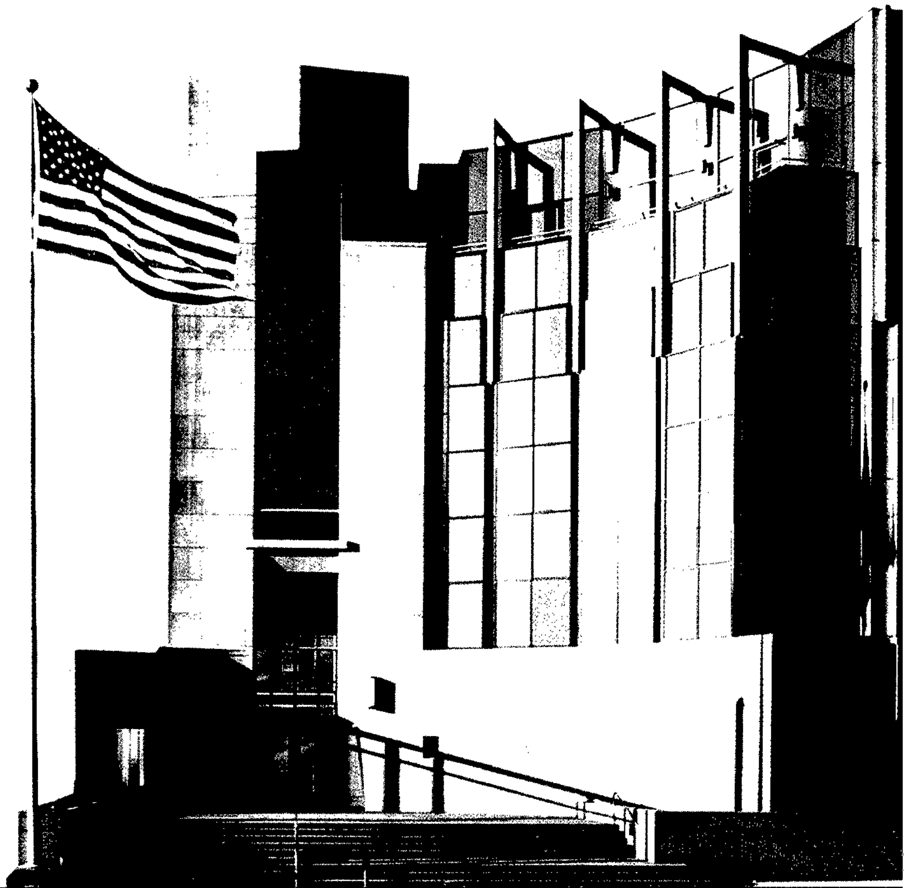
Carnegie Mellon
Software Engineering Institute

Deriving Architectural Tactics: A Step Toward Methodical Architectural Design

Felix Bachmann
Len Bass
Mark Klein

March 2003

TECHNICAL REPORT
CMU/SEI-2003-TR-004
ESC-TR-2003-004



Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

Deriving Architectural Tactics: A Step Toward Methodical Architectural Design

CMU/SEI-2003-TR-004
ESC-TR-2003-004

Felix Bachmann
Len Bass
Mark Klein

March 2003

Architecture Tradeoff Analysis Initiative

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

Christos Scondras
Chief of Programs, XPK

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

List of Figures

List of Tables

Abstract

This is one of several reports that provide the current status on the work being done by the Software Engineering Institute (SEI^{SM}) to understand the relationship between quality requirements and architectural design. The ultimate objective of this work is to provide analysis-based guidance to designers so that the quality attributes of generated designs are more predictable and better understood.

Currently, four distinct problems must be solved to achieve that objective: (1) the precise specification of quality attribute requirements, (2) the enumeration of architectural decisions that can be used to achieve desired quality attribute requirements, (3) a means of coupling one quality attribute requirement to the relevant architectural decisions, and (4) a means of composing the relevant architectural decisions into a design. Embodying the solutions to these four problems into a design method that is sensitive to business priorities is an additional problem. This report deals with the third problem—coupling one quality attribute requirement to architectural decisions that achieve it.

This report provides initial evidence that there is, in fact, a systematic relationship between general scenarios, concrete scenarios, architectural tactics, and design fragments. It examines, in detail, two concrete scenarios—one for performance and one for modifiability—and describes how to move from each scenario, through tactics, to design fragments that satisfy the scenario.

1 Introduction

This is one of a sequence of reports [Bachmann 02, Bachmann 00] that provide the current status of the work being done by the Software Engineering Institute (SEI^{SM}) to understand the relationship between quality requirements and architectural design. Our ultimate objective is to provide analysis-based guidance to designers so that the quality attributes of generated designs are more predictable and better understood.

At this point, we see four distinct problems that must be solved to achieve our objective: (1) the precise specification of quality attribute requirements, (2) the enumeration of architectural decisions that can be used to achieve desired quality attribute requirements, (3) a means of coupling one quality attribute requirement to the relevant architectural decisions, and (4) a means of composing the relevant architectural decisions into a design. Embodying the solutions to these four problems into a design method that is sensitive to business priorities is an additional problem. This report deals with the third problem—coupling one quality attribute requirement to architectural decisions that achieve it—while general scenarios and architectural tactics address the first two problems.

In a prior report [Bass 01], we introduced the concept of general scenarios as a precise system-independent specification of quality attribute requirements. General scenarios provide a structured means of stating quality attribute requirements. System-specific quality attribute requirements—called concrete scenarios—are instances of general scenarios.

We also introduced the concept of *architectural tactic* as a characterization of architectural decisions that are used to achieve a desired quality attribute response. An *architectural tactic* is a means of satisfying a quality-attribute-response measure (such as average latency or mean time to failure) by manipulating some aspect of a quality attribute model (such as performance queuing models or reliability Markov models) through architectural design decisions.

Restating the subject of this report in terms of concrete scenarios and architectural tactics, we are focusing on the derivation of a set of relevant architectural tactics (actually, we go further and derive a set of design fragments) from a concrete scenario. The key idea of our approach is that reasoning frameworks for quality attributes enable this linkage.

[SM] SEI is a service mark of Carnegie Mellon University.

This report provides initial evidence that there is, in fact, a systematic relationship between general scenarios, concrete scenarios, architectural tactics, and design fragments. We examine, in detail, two concrete scenarios—one for performance and one for modifiability—and describe how to move from each scenario, through tactics, to design fragments that satisfy that scenario.

The process of moving from concrete scenarios to design fragments is based on the following set of relations:

- A concrete scenario *is an instance of* a general scenario.
- A general scenario *contains* quality attribute parameters.
- Reasoning frameworks *comprise* independent and dependent quality attribute parameters and their relations.
- A quality attribute model *is an instance of* a reasoning framework.
- Tactics *comprise* reasoning framework rules and architectural design rules.

The process starts with a concrete scenario and then determines the general scenario of which it is an instance. The general scenario contains quality attribute parameters that will point to one (or possibly several) quality-attribute-reasoning framework (e.g., scheduling theory or queuing theory). Reasoning frameworks comprise a set of independent and dependent parameters and their relations and thereby offer the vocabulary and analytic machinery for describing and deducing specific system properties. The concrete scenario specifies (or binds) values for some of the parameters. Other parameters are free in the sense that they have not yet been assigned values; free parameters become the focus of design decisions.

Making design decisions starts with developing a model (such as a scheduling or queuing model). A quality attribute model is an instance of using a quality-attribute-reasoning framework. Given the model, values for the unbound independent parameters are estimated. When all the independent parameters have values, the value of the dependent parameter(s) can be computed. The dependent parameter is then compared with the quality attribute requirement specified by the concrete scenario. Tactics can be used to adjust the value of independent parameters in a manner that will satisfy the concrete scenario. When the requirement is satisfied, tactics can then be used to determine suitable design fragments that are consistent with values of the parameters.

In Section 2, we repeat the key concepts from quality attribute scenarios and tactics. Next, in Section 3, we introduce our sample domain—a garage door opener—and derive design fragments for one performance and one modifiability scenario. Then, in Section 6, we take a step back and provide a general treatment of the steps we exemplified using our two scenarios. Section 7 provides a discussion of some problems that must be solved to compose the design fragments and embed our solutions into a design method. Appendix A provides the general scenarios for performance and modifiability, and Appendix B provides the tactics for those two attributes.

2 Review of Key Concepts

We begin by reviewing three key concepts that we introduced in previous reports [Bachmann 02, Bass 01]: *general scenarios*, *concrete scenarios*, and *tactics*.

2.1 General and Concrete Scenarios

A *general scenario* is a precise system-independent specification of a type of quality attribute requirement that consists of [Bass 03]

- a *stimulus:* a condition that needs to be considered when it arrives at a system
- a *response:* the activity undertaken after the arrival of the stimulus
- a *source of the stimulus:* the entity (e.g., a human or computer system) that generated the stimulus
- an *environment:* the conditions under which the stimulus occurs; for example, when the system is in an overload condition
- a *stimulated artifact:* Some artifact is stimulated. It could be the whole system or pieces of it.
- a *response measure:* the attribute-specific constraint that must be satisfied by the response

An example of a general scenario is

> *A periodic event from an independent source arrives at the system under normal conditions. The system has to process the stimulus within certain latency.*

The general scenario provides a template for a class of requirements. Appendix A discusses general scenarios in more detail.

Ultimately, general scenarios must be changed into system-specific *concrete scenarios*. To make this change, the generic vocabulary is replaced with concrete, system-specific vocabulary. For example, the performance scenario shown above can be made system specific as follows:

> *An event from sensor X arrives every 10 milliseconds (ms) at a system that operates under normal conditions. The system has to process the stimulus within 1 ms.*

System-specific quality attribute requirements can be specified using concrete scenarios.

2.2 Tactics

An *architectural tactic* is a means of satisfying a quality-attribute-response measure by manipulating some aspect of a quality attribute model through architectural design decisions. This definition of tactic is different from (and we hope clearer than) our previous definition [Bachmann 02]. We changed the definition to emphasize our reliance on quality attribute models since that is key in this report.

The definition has the following consequences:

- An architectural tactic bridges the quality attribute model and the architectural design. It does so by specifying how the parameters of a quality attribute model (the input, the independent variables, and the properties of the model's elements) can be controlled through architectural decisions to achieve a desired response measure. This means that an architectural tactic represents codified knowledge about the relationship between architectural decisions and quality attribute parameters.
- An architectural tactic uses knowledge from various reasoning frameworks (such as scheduling theory and queuing theory) to operationalize the creation of quality attribute models (such as scheduling and queuing models) that mirror the architecture being designed. As such, when we use the term *quality attribute model*, we mean an instance of using a reasoning framework to predict and control the quality attribute behavior of the developing architecture.

At this time, we do not have a complete outline of a tactic description. That will be included in a later report. Nevertheless, throughout this report, we point out important information that needs to be captured in a tactic description.

We discuss performance and modifiability tactics in more detail in Appendix B. An example of a performance tactic is

> *Reduce the computational overhead.* Computational work usually requires operating-system (OS) and middleware services to manage process interaction, communications, and the like. One important parameter in many performance models (such as queuing and scheduling models) is execution time—one source of which is overhead. The purpose of this tactic is to highlight one place where execution time is manifested in architectural designs.

An example of a modifiability tactic is

> *Hide information.* This tactic is based on dividing a module's responsibilities into two categories: public and private. Public responsibilities are those that are visible from both inside and outside the module. Private responsibilities are those that are visible only from inside the modules. The goal of this tactic is to limit the public responsibilities and make them visible through an interface. A parameter of a modifiability model is the dependency relations among the modules. The purpose of this tactic is to highlight one technique for breaking some dependency relations.

Tactics are not necessarily independent. The application of one tactic may assume that another tactic has already been applied. For example, the application of the *Determine the appropriate scheduling strategy* tactic assumes that logical concurrency exists in the system. The application of one tactic may also require additional tactics to be applied. For example, applying the *Break the dependency chain* tactic to insert an intermediary would likely require additional tactics to isolate that intermediary's responsibilities.

3 Garage Door Example

Our sample design problem is that of a garage door opener. The controller for a garage door opener is an embedded real-time system that reacts to "open" and "close" commands from several buttons installed in the house and from a remote control unit, usually located in a car. The controller then controls the speed and direction of the motor, which opens and closes the garage door. The controller also reacts to signals from several sensors attached to the garage door. One of those sensors detects resistance to the door's movement. If the amount of resistance measured by this sensor is above a certain limit, the controller interprets that resistance as an obstacle between the garage door and the floor. As a reaction, the motor closing the garage door is stopped.

Many scenarios specify the requirements for the controller software. For the remainder of this report, we focus on two specific concrete scenarios—the performance and modifiability scenarios shown below.

1. Performance scenario: If an obstacle (person or object) is detected by the garage door during descent, the door must halt within 0.1 seconds.
2. Modifiability scenario: The processor (central processing unit [CPU]) used in different products differs. Adaptation of the software to those different processors should be done within one person-day.

In the following sections, we show the relation of the above concrete scenarios to a collection of design fragments that supports the achievement of the desired response measures. Each derivation of a concrete scenario to a design fragment follows the steps below. In Sections 4 and 5, these steps are described for use with performance and modifiability scenarios, respectively.

1. Pick a concrete scenario.
2. Type-check the concrete scenario.
3. Identify candidate reasoning frameworks.
4. Determine the bound and free parameters.
5. Determine the tactics associated with the free parameters.
6. Assign the free parameters an initial set of values.

7. Select tactics and apply them to the bound values to achieve the required response measure.

8. Allocate responsibilities to the architectural elements of the design fragments that are associated with the selected tactics.

Steps 1 and 2 are scenario dependent and provide information for subsequent steps. Steps 3, 4, and 5 are intended to set up the model in the reasoning framework; they identify the parameters that can be manipulated to satisfy the scenario response. Steps 6 and 7 are intended to solve the model set up in Steps 3, 4, and 5. Step 8 translates the tactics used into design fragments.

4 Scenario 1: Performance Scenario to a Design Fragment

Our starting point is the following performance scenario:

> If an obstacle (person or object) is detected by the garage door during descent, the door must halt within 0.1 seconds.

4.1 Step 1: Pick a Concrete Scenario

These steps are intended to be applied in the context of a design method. One thing the design method must specify is some procedure for choosing the scenario or scenarios for which design fragments are to be derived next. For this report, we assume this is the only scenario to consider.

4.2 Step 2: Type-Check the Concrete Scenario

Often, the concrete scenario is not well formed (by our definition of general scenario). Many items are implicit in a particular context and will not be presented in a requirement. Type-checking means ensuring that the concrete scenario contains all the parts of a well-formed scenario and that each part has a valid value as defined by the general-scenario-generation tables in Appendix A. Two important outcomes of type-checking a concrete scenario are (1) identifying the general scenario of which the concrete scenario is an instance and (2) identifying an initial set of responsibilities.

To determine the associated general scenario, we divide the concrete scenario into its constituent parts. The parts of the concrete scenario we're using are listed in Table 1.

Table 1: Parts of the Sample Concrete Performance Scenario

Part Name	Value
source	not defined
stimulus	obstacle (person or object) is detected
environment	during descent
artifact	by the garage door system
response	The door must halt.
response measure	within 0.1 seconds

The values given for the scenario parts map to the values provided by the general performance scenarios. One value is the arrival distribution, which describes how often an obstacle is detected. Because we can assume this happens infrequently, there is a bound on how frequently it can occur (known as a sporadic arrival distribution). Another parameter is the deadline, which is .1 seconds from when the obstacle is detected.

Comparing the values, we can derive the general scenario described in Table 2.

Table 2: Parts of the Derived Performance General Scenario

Part Name	Value
source	one of a number of independent sources
stimulus	sporadic events arrive
environment	normal conditions
artifact	system
response	processes stimuli
response measure	deadline

Note that the general scenario provides the type of information, while the concrete scenario provides the values. Using the general scenario, we could also assign a value to the source part, which was not described in the concrete scenario. The source would be something like "the obstacle-detection sensor."

Casting the quality attribute requirement as a concrete scenario provides a natural method for identifying some of the initial responsibilities of the garage door opener. Performance general scenarios help the user to determine the following three responsibilities:

1. detect <from a stimulus>
2. determine <from the environment>
3. carry out <from the response>

These responsibilities come into play in Step 8: "Allocate responsibilities to the architectural elements of the design fragments that are associated with the selected tactics."

In our example, the responsibilities of the garage door opener are to

- Detect the obstacle.
- Determine that the garage door is descending.
- Halt the garage door.

4.3 Step 3: Identify Candidate Reasoning Frameworks

For modifiability and performance, we intend to construct tables such as Table 3 below. For each response measure, Table 3 identifies a reasoning framework (or possibly frameworks) that will be useful for satisfying the response measure. The table identifies important parameters of the reasoning framework. For example, given "hard deadline" as a response measure, the table tells us that scheduling theory will be useful. It also tells us that scheduling theory uses the following independent parameters to determine the dependent parameter for worst-case latency: execution times, arrival distributions, the number of units of concurrency, the priority of each unit of concurrency, and the number of processors. Worst-case latency is compared with the deadline to determine if that type of response measure (i.e., deadlines) is satisfied. This table can also help to determine which tactics are applicable for a particular scenario; recall that the purpose of tactics is to manipulate the independent parameters in a manner that will satisfy the response measure.

Table 3: Performance Reasoning Frameworks and Their Parameters

Reasoning Framework	Quality Attribute	Independent Parameters	Dependent Parameter	Response Measure
Scheduling theory	Performance	• Execution times (with relatively low variability) of units of concurrency • Arrival distribution (a.k.a., period, with relatively low variability) of units of concurrency • Number of units of concurrency • Priority of each unit of concurrency • Number of processors	Worst-case latency	Hard-deadline requirement
Queuing theory	Performance	• Execution-time distribution associated with servicing event streams • Arrival distribution of event streams • Number of event streams • Number of servers	Average-case latency Average queue depth	Soft-deadline requirement Data loss requirement

We know from looking at our concrete scenario that we have sporadic event arrivals and a hard-deadline requirement. That requirement suggests that the applicable reasoning framework is likely to be scheduling theory. Sporadic arrivals are arrivals that cannot occur arbitrarily and often. They indicate a bound on the arrival rate variability, again showing that scheduling theory might be relevant. The other relevant parameters include execution time, the number of units of concurrency, the priority of each unit, and the number of processors. The response measure is the deadline.

We have not yet considered the problem of choosing between several suitable reasoning frameworks. Although we suspect that if several reasoning frameworks have the same independent and dependent parameters, they are probably equivalent.

4.4 Step 4: Determine the Bound and Free Parameters

In this step, we determine the bound and free parameters of the applicable reasoning framework. Scheduling theory is concerned with calculating the worst-case latency associated with carrying out each scenario, given the execution time, arrival period, and priority associated with each unit of concurrency; the number of units of concurrency; and how each unit is allocated to one or more processors.

From our type-checking, we know that the arrival distribution is sporadic with an occurrence of one (within some reasonable time bound), and so the arrival distribution parameter is fixed. Design constraints can also bind parameters. For example, the business case of the developer of the garage door opener's software might dictate that only a single processor is used in the garage door. Thus, the "number of processors" parameter is also fixed.

To summarize, bound parameters include

- arrival distribution
- number of processors

And free parameters include

- number of units of concurrency
- priority of each unit of concurrency
- execution time of responsibilities

4.5 Step 5: Determine the Tactics Associated with the Free Parameters

The objective of this step is to determine which tactics are applicable to the free parameters identified in the previous step. We intend to construct tables such as Table 4 that relate independent parameters of a reasoning framework to the tactics that affect them.

Table 4: *Tactics That Control Parameters for the Scheduling Theory's Reasoning Framework*

Parameter	Tactic That Affects It
execution times	**Tactics for Managing Demand** Reduce computational overhead Bound execution times Control the demand for resources Increase the computational efficiency of algorithms
arrival distribution	**Tactics for Managing Demand** Manage the event rate Control the frequency of sampling external events Bound queue sizes
number of units of concurrency	**Tactics for Arbitrating Demand** Increase logical concurrency Determine appropriate scheduling policy
priority of each unit of concurrency	**Tactic for Arbitrating Demand** Determine appropriate scheduling policy
number of processors	**Tactics for Managing Multiple Resources** Increase physical concurrency Balance resource allocation

We also assume that a decision procedure is associated with each reasoning framework. This procedure is a set of rules governing which tactics are relevant in particular situations. Table 5 through Table 7 provide a sampling of the rules in a decision procedure for scheduling theory.[1] The decisions are based strictly on which parameters are considered to be bound and which are considered to be free:

1. Which parameters are bound?
   - arrival distribution: Arrivals are infrequent.
   - number of processors: A single processor is being used.

   From the first rule in Table 5, we conclude that the bounded arrival distribution rules out the following tactics: *Manage event rate* and *Control the frequency of sampling external events*.

   The single-processor constraint rules out the following tactics: *Increase physical concurrency* and *Balance resource allocation.*

2. Which parameters are free?
   - execution time: The responsibilities will suggest a likely range, but this is not yet fixed.

[1] An important aspect of this ongoing work is to flesh out the set of rules.

- number of units of concurrency: The value of this parameter will be determined later.
- priority of each unit of concurrency: The value of this parameter will be determined later.

The following tactics are concerned with manipulating execution time: *Reduce computational overhead, Increase computation efficiency, Control the demand for resources*, and *Bound execution time.*

Three rules from our performance decision procedure that are applicable for this step are shown in Table 5.

Table 5: Example Rules from Our Performance Decision Procedure

- If the arrival distribution is bounded, the *Manage event rate* and *Control the frequency of sampling external events* tactics cannot be used to control worst-case latency.
- If execution time is a free parameter, consider using the following tactics: *Reduce computational overhead, Increase computation efficiency, Control the demand for resources*, and *Bound execution time.*
- If the number of processors is bound, eliminate the following tactics as candidates: *Increase physical concurrency* and *Balance resource allocation.*

4.6 Step 6: Assign the Free Parameters an Initial Set of Values

Two things occur during this step. First, the architect offers his/her best guess for values of the free parameters. The list of applicable tactics suggests factors that impact the setting of these values. Second, the decision procedure rules call attention to possibly problematic situations.

From the previous steps, we know that three of the tactics are relevant to estimating execution time: *Reduce computational overhead, Improve computational efficiency*, and *Bound execution time.* The architect might guess that the sum of the sensor's execution time for physically detecting an obstacle, plus the execution time of the three responsibilities associated with obstacle detection and halting is about 5 ms. The *Bound execution time* tactic calls our attention to the effects of execution-time variability. However, we predict that these responsibilities have very little variability. The *Improve computational efficiency* tactic calls our attention to the particular algorithm used to detect that an obstacle exists. The *Reduce computational overhead* tactic calls our attention to various sources of overhead that represent extra execution time. It is conceivable that each responsibility mentioned above (i.e., detect obstacle, determine that the garage door is descending, and halt the garage door) incur some OS overhead for some preselected real-time OS. Consequently, we estimate that the OS adds an additional 1 ms of overhead. Since this is the only scenario being considered, we can assume that this scenario's responsibilities are all allocated to a single unit of concurrency. This assumption may change in the future as we consider multiple scenarios and deal with already-made design decisions—but for now, our assumption is adequate.

While we don't yet know all the details of the other responsibilities in the system, we do know that other responsibilities with associated execution times will exist. If we're not careful, those responsibilities could potentially have an adverse effect on the ability of this scenario to be realized. We are not yet ready to assign values to their associated execution times.

The second consideration at this stage is to examine the scenario to determine if it is unreasonable or problematic. For example, if execution times or arrival rates vary considerably and deadlines can never be missed, problems might arise. Examples of rules that call attention to such potentially problematic situations are shown in Table 6. However, none of these situations applies to the current scenario.

Three rules from our performance decision procedure that are applicable for this step are shown in Table 6.

Table 6: Example Rules That Apply to Step 6

- If the scenario has a hard-deadline-response requirement that cannot be relaxed, and if arrivals can occur arbitrarily close to one another, use one of the following tactics to ensure a lower bound on the interarrival interval: *Manage event rate* and *Control sampling frequency.*
- If the scenario has a hard-deadline-response requirement that cannot be relaxed, and if execution times vary considerably to the point that they can approach or exceed the hard deadline, consider applying the *Bound execution time* tactic.
- If either of the above "unbounded" conditions applies, but arrival rate and execution time are bound parameters, declare the requirement untenable.

4.7 Step 7: Select Tactics and Apply Them to the Bind Values to Achieve the Required Response Measure

At this point, all the parameters have values, and we have a candidate list of applicable tactics. The first thing we need to do is to calculate the value of the dependent parameters, given the current values of the independent parameters, to determine whether the response measure can be met.

Without considering the effects of other responsibilities, the model in this case is fairly simple. The only contributors to latency are execution time and overhead, 5 ms and 1 ms, respectively. Their sum is well under the deadline of 100 ms (i.e., .1 seconds), leaving 94 ms to spare. On the other hand, it is very conceivable that the other responsibilities in the system combined take more than 94 ms.

Four rules from our performance decision procedure that are applicable for Step 7 are shown in Table 7. The last rule in Table 7 suggests that the design decisions we make in Step 8 be consistent with our simple model—that is, they should ensure that the latency associated with this scenario's responsibilities is not affected by any other responsibilities.

Table 7: Example Rules That Apply to Step 7

- If the execution time associated with the arrival is close to the deadline, consider reducing the execution time by using the following tactics: *Reduce overhead, Bound execution time,* and/or *Increase computation efficiency.*
- If the difference between the worst and best case is significant, review the following tactics and apply their modeling techniques to assess the miss rates and average latency, respectively: *Bound execution times* and/or *Bound queue sizes.*
- If the response requirement for all scenarios can be achieved even with the worst-case delay due to the other responsibilities of the other scenarios, do one of the following:
  - Allocate responsibilities to one of the existing units of concurrency using the *Offline scheduling* tactic.
  - Allocate responsibilities to a new unit of concurrency using the *Increase logical concurrency* and *Time-based scheduling* tactics.
- If the current scenario cannot suffer the worst-case delay due to the other responsibilities, consider them to be time sensitive and use the following tactics to create an appropriate scheduling policy: *Offline scheduling, Time-based scheduling* (such as deadline monotonic scheduling), and/or *Increase logical concurrency.*

The next thing we need to do is to look at one or more of the applicable tactics to determine which parameters can be adjusted to satisfy the concrete scenario.

The relevant tactics entering into this step include

- Controlling resource demand through the *Control the demand for resources* and *Bound execution time* tactics assumes that multiple events are occurring. Our scenario is only concerned with a single event (i.e., obstacle detected).
- The *Reduce computational overhead* and *Increase computational efficiency* tactics both affect the latency. The computational overhead for this scenario comes from the OS choice. Unless the scenario is unachievable (which is not yet the case), we do not want to change that choice. Increasing the computational efficiency of the obstacle detection al-

gorithm would save possibly 1 ms. Applying the reasoning framework with this new execution-time value shows that the execution time does not have much impact.[2]

- The *Increase logical concurrency* and *Determine scheduling policy* tactics together enable the satisfactory achievement of the response measure. For this reason, we carry these tactics forward into the next step.

Up to now in this step, tactics have been used to set and/or adjust model parameters to satisfy the current concrete scenario's response measure. However, the scenario might pose an untenable requirement, or the collection of scenarios considered up to this point might be untenable in aggregate. If either situation occurs, tactics should offer some ideas for how to relax the requirements or design constraints.

Two rules from our performance decision procedure that are useful for identify and relaxing requirements and/or design constraints are shown in Table 8.

Table 8: Rules for Relaxing Requirements and/or Design Constraints

- If the response requirement is specified as a hard deadline, but limited misses can be tolerated, recharacterize deadlines as follows:
  - firm deadlines: Completing the responses before the deadline is very important. Missing the deadline can be tolerated occasionally. A specific bound on the miss rate needs to be specified.
  - soft deadlines: In this case, the term *deadline* is a misnomer. The average-latency requirement needs to be specified.
- If the time-sensitive set of responsibilities is not schedulable, incorporate a notion of importance-based scheduling to handle overload situations using the *Semantic-importance-based scheduling* tactic, or add more resources using the *Increase physical concurrency* tactic.

4.8 Step 8: Allocate Responsibilities to the Architectural Elements of the Design Fragments That Are Associated with the Selected Tactics

We assume that a design fragment is associated with each tactic. The fragment associated with the *Increase logical concurrency* and *Determine scheduling policy* tactics is shown in Figure 1. This design fragment shows two threads, one of which has a higher priority than the

[2] This example points out the importance of understanding how sensitive the reasoning framework is to the various parameters. For example, knowing the impact of halving the value of a parameter tells the architect how much effort to spend on that type of improvement. If the impact is small, little effort is justified; if the impact is large, more effort is justified.

other. The fragment rules call attention to the existence of a scheduler (not shown in Figure 1) whose responsibilities must be allocated to some module in the system.

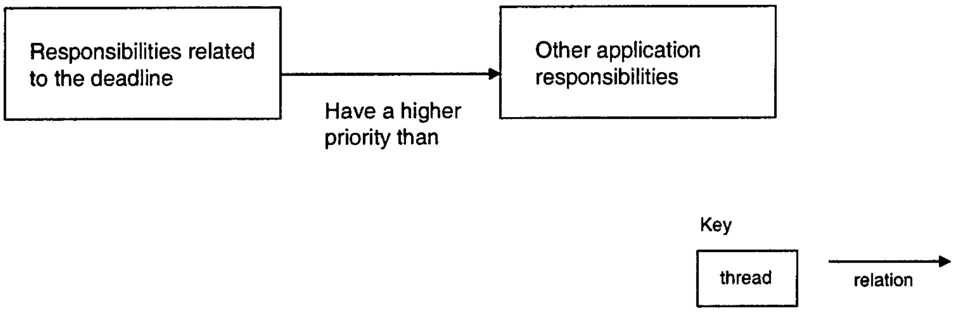

Responsibilities related to the deadline
Have a higher priority than
Other application responsibilities
Key
thread
relation

Figure 1: Design Fragment Associated with the Increase Logical Concurrency *and* Determine Scheduling Policy *Tactics*

The generic responsibilities in the design fragment shown in Figure 1 are instantiated with the values derived from type-checking and result in the design fragment shown in Figure 2.

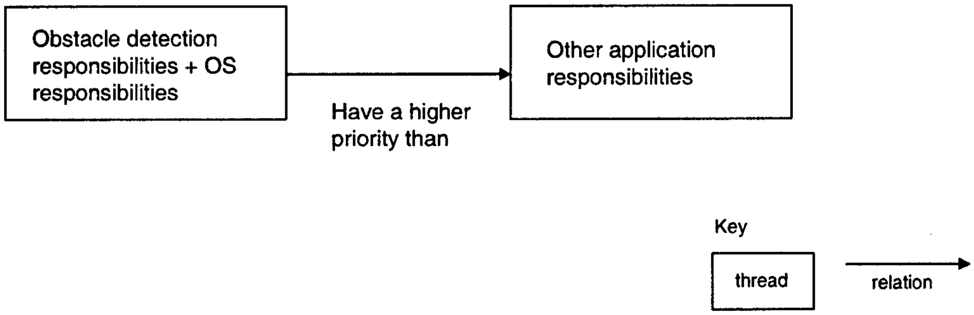

Obstacle detection responsibilities + OS responsibilities
Have a higher priority than
Other application responsibilities
Key
thread
relation

Figure 2: Design Fragment from Figure 1 Instantiated for a Concrete Scenario

Observe the relationship between the design fragment and the associated analysis model. The model states that obstacle detection responsibilities must be scheduled with a priority higher than that of other responsibilities. The model also identified the OS responsibilities as a portion of the obstacle detection responsibilities.

5 Scenario 2: Modifiability Scenario to a Design Fragment

Modifiability is the degree to which an architectural design can be modified easily in the future. We want to design an architecture that is prepared for certain modifications.

To determine the possible tactics for modifiability, we follow the same steps listed in Section 3 and assume that we are designing based solely on the modifiability scenario.

5.1 Step 1: Pick a Concrete Scenario

Our modifiability scenario is

> The processor (CPU) used in different products differs. Adaptation of the software to the different processors should be done within one person-day.

This scenario describes a requirement that typically occurs when building a product line. By *processor*, we mean only the physical hardware and not the software platform associated with it. Any platform considerations will be included in the software we design.

5.2 Step 2: Type-Check the Concrete Scenarios

In this step, we map the concrete scenario to the associated general scenario for modifiability. To create that general scenario, we divide the concrete scenario into its constituent parts as shown in Table 9.

Table 9: Parts of the Example Concrete Modifiability Scenario

Part Name	Value
source	not defined
stimulus	change processors (CPUs)
environment	not defined
artifact	garage door system
response	adaptation to different processors
response measure	within one person-day

The values given for the scenario parts map to the values provided in the general-scenario-generation table (see Appendix A). Comparing the values and filling in the unspecified fields, we can derive the general scenario shown in Table 10.

Table 10: Parts of the Example General Modifiability Scenario

Part Name	Value
source	developer
stimulus	wants to add/delete/modify/change functionality
environment	compile time
artifact	system
response	makes modification without affecting other functionality
response measure	effort

The concrete scenario mentions two sets of responsibilities.[3] One set includes all the responsibilities assigned to the processor (from the stimulus table entry); the other includes all the responsibilities of the software (from the artifact table entry). The responsibilities of the processor will change, and we have to make sure that the effect on the responsibilities of the software is within the limits stated in the concrete scenario (one person-day). We represent this relationship in Figure 3.

[3] We treat the processor as though it implements certain responsibilities. Although this may seem somewhat strange and blurs the line between hardware and software, it allows us to treat hardware modifications as software modifications.

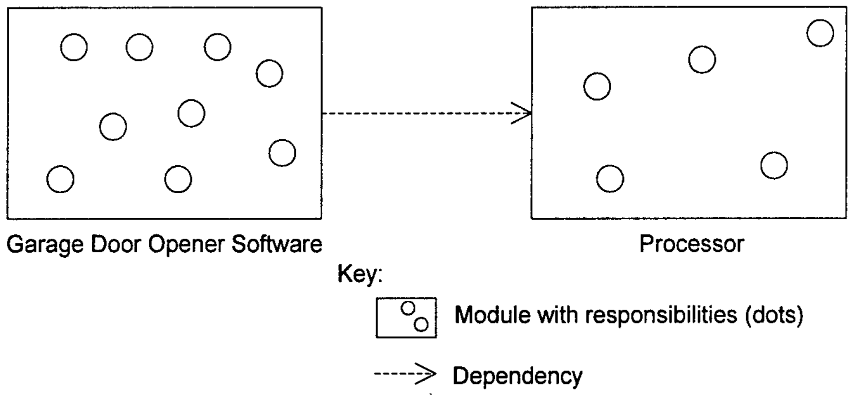

Garage Door Opener Software
Processor
Key:
Module with responsibilities (dots)
Dependency

Figure 3: Processor-Dependent Responsibilities

5.3 Step 3: Identify Candidate Reasoning Frameworks

The reasoning frameworks for modifiability are not as precise as those for performance. The important question the reasoning framework needs to answer is "how much will it cost to make a certain change?"

The answer to this question is typically stated in terms of *effort* or *time*. In this case we are concerned with return on investment. Preparing an architecture to support a certain type of change requires an investment in terms of effort. The following basic formula shows what should be achieved:

$$e_i \leq e_s \ with \ e_s = n(e_n - e_w)$$

where

- e_i is the effort to prepare the architecture for a certain change (investment).
- e_s is the effort saved because the architecture was prepared for the change.
- e_n is the effort used to make a change for which the architecture was not prepared.
- e_w is the effort for making the change in the prepared architecture.
- n is the number of times this type of change occurs during the lifetime of the architecture.

Basically, the relation states that the savings should be greater than or equal to the investment. The efforts for making a change (e_n and e_w) depend on several factors, such as effort for

- finding affected responsibilities
- adapting the responsibilities

- testing all the responsibilities within a changed module
- testing all the responsibilities of the modules that depend on the adapted module

The complete effort E for making this type of change n times during the lifetime of the software, given that the architecture is prepared for this type of change, is the sum of the efforts required for every change plus the investment efforts:

$$E = e_i + \sum_{i=1}^{n} e_{wi}$$

A particular organization may choose to be influenced by either of the two formulae presented. The initial formula suggests that architecting software to be modifiable may be worthwhile only if this up-front cost can be amortized over some number of future modifications. The second formula suggests that it is important to track the total cost of modification. If an organization wants to stay in a particular market, it may not matter how expensive a change is. If an organization has both limited developers and limited time to react to a change request (e.g., from an important customer), the initial investment may not be as important as the cost (time) of any particular change.

Table 11 shows the main parameters influencing the efforts of making a change.

Table 11: Modifiability Reasoning Framework and Parameters

Reasoning Framework	Quality Attribute	Independent Parameters	Dependent Parameters	Response Measure
Dependency analysis	Modifiability	• number of primary modules affected by the change • number of responsibilities within the primary modules affected • probability that publicly visible responsibilities of a primary module are affected • number of secondary modules that depend on the changed publicly visible responsibilities of a primary module • number of responsibilities within the secondary modules affected	Effort for making a change (e_w)	Cost constraint

Figure 4 illustrates those parameters. A change affects several responsibilities in one or more modules. If publicly visible responsibilities are affected, most likely one or more other responsibilities in one or more other modules are affected. Keep in mind that each affected responsibility comes with the costs of finding, changing, and testing it.

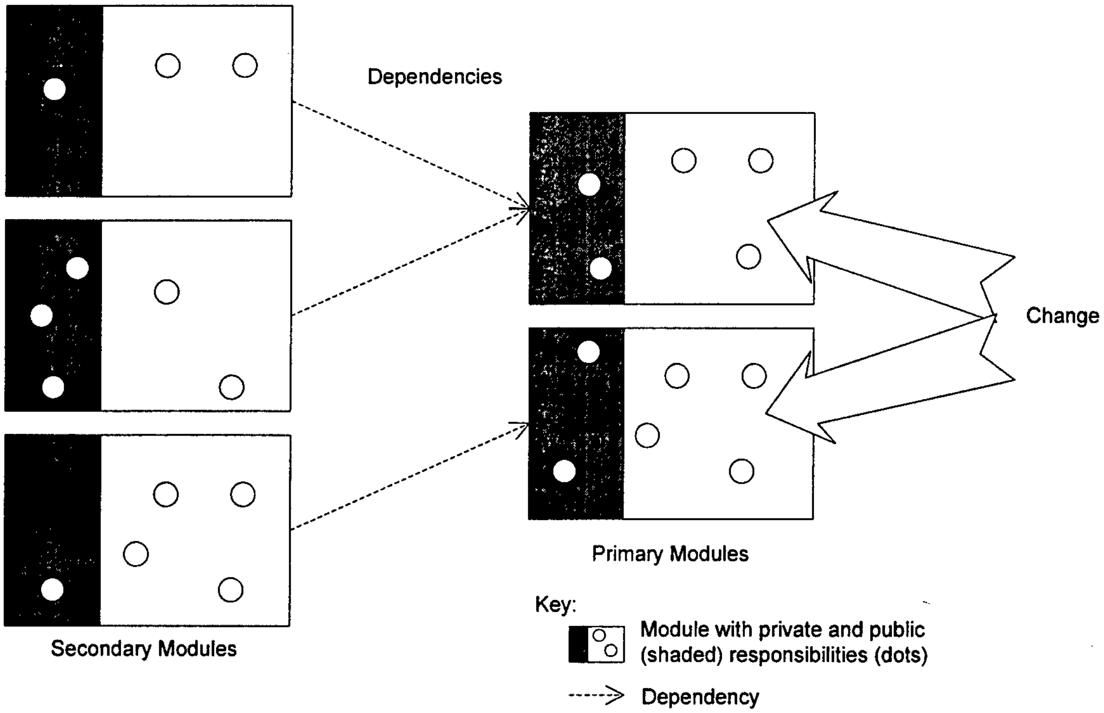


Figure 4: Parameters That Influence the Effort of a Change

If secondary modules are affected by a change, this list has to be applied recursively, where the secondary modules become primary modules.

5.4 Step 4: Determine the Bound and Free Parameters

Since we are only considering one scenario, the reasoning framework contains the responsibilities as shown in Figure 3. We consider two modules—the processor and the garage door opener software—which both depend on the processor. The modifiability scenario states that the processor will be replaced by another one, meaning that potentially all the processor's responsibilities are changed.

An examination of the differences between an existing processor and the new processor will reveal the responsibilities, especially the publicly visible ones that will be added, removed, or modified when using the new processor. The result of the examination binds the number of responsibilities affected in the processor.

Examining the affected responsibilities if they are used by the garage door opener software determines the ratio between public and private responsibilities and therefore binds the "probability that publicly visible responsibilities are affected" parameter. In our case, we set the value of that parameter to 1 because we know that changing the processor will affect at least 1 visible responsibility. We therefore consider this parameter to be fixed. Furthermore, since all the responsibilities of the processor that affect the software are public, the probability that publicly visible responsibilities of the processor are affected is 1.

The "number of primary modules affected" parameter is also set to 1. The garage door opener will be a single-processor application.

Therefore, we have the following values for these bound parameters:

- primary modules affected by the change (1—the processor)
- responsibilities changed (processor responsibilities)
- probability that publicly visible responsibilities of the processor are affected (1)

And we have the free parameter "number of secondary modules that depend on the changed publicly visible responsibilities" for which a value must be assigned.

5.5 Step 5: Determine the Tactics Associated with the Free Parameters

From the parameters listed above and in Table 11, we see that the following tactics are associated with the free parameters:

- Limit options
- Break the dependency chain
- Limit communication paths
- Make data self-identifying

The last two tactics—*Limit communication paths* and *Make data self-identifying*—are not feasible given that we have no control over the processor hardware.

Furthermore, the *Break the dependency chain* tactic involves introducing an intermediary, which may, in turn, require restricting the visibility of the intermediary's responsibilities using the following tactics:

- Hide information
- Raise the abstraction level

- Maintain existing interfaces
- Separate the interface from the implementation

Using any of the tactics mentioned in Table 12 contributes to the investment costs e_i. Only the *Limit options* tactic would actually limit the investment costs because it reduces the number of changes the architecture must be able to accommodate.

Table 12: Summary of Tactics and the Parameters They Influence

Parameter	**Tactic That Affects It**
number of occurrences of a change	**Tactics for Localizing Expected Modifications** Limit options (The fewer the possible options, the less often a change occurs.)
number of primary modules affected by a change	**Tactics for Localizing Expected Modifications** Limit options (The fewer the possible options, the less often a change occurs and the less modules are affected.) Isolate the expected change (Separate what is likely to change from parts that are unlikely to change.) Maintain semantic coherence (Put together parts that have strong dependencies on each other so they are easier to find.)
number of responsibilities of the primary modules affected by the change	**Tactics for Localizing Expected Modifications** Abstract common services in the primary modules (Only a limited number of services is affected.) Limit options (The fewer the possible options, the fewer responsibilities are affected.)
probability that a change to a primary module becomes visible outside that module	**Tactics for Localizing Expected Modifications** Limit options (The fewer possible the options, the fewer responsibilities are affected.) Raise the abstraction level (The more abstract a service is, the more possible it will be that changes can be supported invisibly.) **Tactics for Restricting the Visibility of Responsibilities** Hide information (The more responsibilities are private, the less likely a change becomes visible.) Maintain existing interfaces (Changes become visible only through new interfaces.) Separate the interface from the implementation (New behavior can be introduced with a low probability of making it visible.)
number of secondary modules affected by a publicly visible change	**Tactics for Preventing the Ripple Affect** Break the dependency chain (Intermediaries of different types block different types of dependencies from propagating.) Make the data self-identifying (Tagging data with syntax information enables secondary modules to adjust automatically.) Limit communication paths (The fewer the dependencies, the fewer secondary modules are affected.)

5.6 Step 6: Assign the Free Parameters an Initial Set of Values

In our example, we must set initial values for the

- number of secondary modules that depend on the changed publicly visible responsibilities of a primary module
- number of responsibilities in the secondary module(s) that are affected

The number of secondary modules affected is set to 1. However the number of affected responsibilities in the secondary module is likely to be significant. With those initial values, e_w will most likely be much higher than the required cost constraint (one person-day).

5.7 Step 7: Select Tactics and Apply Them to the Bind Values to Achieve the Required Response Measure

We now apply the tactics from our list of candidate tactics (see Section 5.5) until we achieve a satisfying result. The procedure is based on the set of rules shown in Table 13. These rules guide the use of modifiability tactics.

In this case, the *Limit options* tactic asks whether all possible processors are to be supported. If they can be limited to a subset (such as those that belong to the same family of processors), designing for change is much simpler, but involves trading off possible market share.

In our example, restricting processors to be in the same family might reduce n to ≤ 5. It also might reduce the probability that a change in the processor becomes visible to the software. Let us assume that in our example only every other supported processor has different publicly visible responsibilities. This means that at least in every other case we can keep the efforts e_w lower than requested by the scenario.

Since we do not change the responsibilities of the processor (because we buy it), we skip rules 2, 3, and 4 because they require changing the responsibilities of the primary affected modules. We therefore proceed to rule 5.

Table 13: Rules That Guide the Use of Modifiability Tactics

1.	If the number of occurrences of the change can be modified, apply the *Limit options* tactic.
2.	If it is impossible to achieve the change within the specified limits when only considering the primary affected modules, apply the following tactics: • Maintain semantic coherence • Isolate the expected change
3.	If the change can't be achieved within the specified limits when only the primary affected modules are considered and after the tactics in rules 1 and 2 have been applied, it is an unsolvable problem.
4.	If the effort required to adapt the affected secondary modules is not within the specified limits and the primary modules can be changed, apply the following tactics to the affected primary modules: • Abstract common services in the primary modules • Raise the abstraction level • Hide information • Maintain existing interfaces • Separate the interface from the implementation
5.	If the effort required to adapt the affected secondary modules is not within the specified limits, apply the following tactics: • Break the dependency chain • Make the data self-identifying • Limit communication paths
6.	If it is impossible to achieve the change within the specified limits after all tactics have been applied, it is an unsolvable problem.
7.	If the investment costs of using the tactics are lower than the possible gain, reconsider the requirement.

Rule 5 leads to use of the *Break the dependency chain* tactic. We therefore have to determine which type of dependencies between the primary and secondary modules should and should not be allowed. Between the garage door opener software and the processor, we have the following dependencies:

1. data syntax and semantics: The language required to instruct the processor to execute is data for the processor. Using an intermediary means the language used is not assembler language.
2. service syntax and semantics: The garage door opener also uses services provided by the processor, such as memory management, input/output (I/O) handling, timer management, and so forth.

3. sequence of data and control: The software depends on the sequence in which the instructions (data) are executed by the processor. It may also depend to some degree on the sequencing of control within the processor (e.g., interrupt handling).
4. interface identity: The software is loaded onto the processor in which it executes and therefore does not have to know the interface identity of the processor.
5. runtime location: The software is loaded onto the processor in which it executes and therefore does not have to know the processor's location.
6. quality of service: This dependency might exist for services the processor provides (e.g., I/O services, processor speed).
7. existence: The software depends on the existence of the processor.
8. resource behavior: The software depends on how the processor executes the instruction, especially timing behavior.

To achieve the required efforts stated in the concrete scenario, almost all the dependencies have to be broken. It is very unlikely that the software will not be loaded on the processor in which it executes. Therefore, the interface identity, runtime location, and existence of the processor will not change, and we will not try to break those dependencies. We also assume that a resource behavior dependency will always exist on the processor.

5.7.1 Using a Compiler as an Intermediary

We can break the data syntax dependencies by introducing a higher language than assembler language. We assume that a compiler for the processor and language we chose exists. The compiler also alleviates the semantic dependencies because it supports the use of a more abstract language and breaks the dependency on data sequencing.

Usually, a compiler also comes with a runtime environment that acts as an intermediary for some services provided by the processor. Standard I/O services, memory management, and time management are examples of services provided by a runtime environment.

This breaks the service syntax dependencies to the supported processor services. Also, the semantic dependencies are reduced because the runtime environment usually provides a more generic interface to the services.

5.7.2 Using an OS as an Intermediary

We still have dependencies in our software on services not concealed by a compiler. Examples include I/O with special devices that manage special memory such as flash memory. To break those dependencies, we use an intermediary that is or acts as an OS. This intermediary breaks the service syntax and resource behavior dependencies on the remaining processor services, as well as reduces the semantic dependencies.

When defining the responsibilities of the intermediary, using the *Raise the abstraction level* and *Hide information* tactics provides a more abstract interface to the processor's services and hides all processor-dependent details.

When use of the rules in Table 13 reveals that the problem is unsolvable, the only way to come to a solution is to change one or more of the parameters that were fixed by the scenarios. Perhaps some of the requirements are not as important as stated, or the acceptable efforts for doing the change can actually be higher than previously thought.

After one or more fixed parameters are changed, the rules can be applied again, and hopefully a possible solution can be found.

5.8 Step 8: Allocate Responsibilities to the Architectural Elements of the Design Fragments That Are Associated with the Selected Tactics

The *Break the dependency chain* tactic has the design fragment shown in Figure 5.

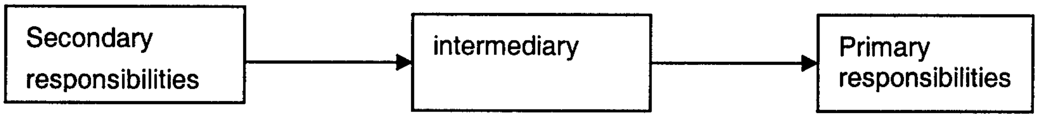


Figure 5: Design Fragment of the Break the Dependency Chain *Tactic*

Applying this design fragment three times and assigning the responsibilities according to the tactics used lead to the design fragment shown in Figure 6. Most of the direct dependencies are now broken. The remaining dependencies of the garage door opener on the processor (e.g., location or existence) are those which probably won't be affected by changing the processor.

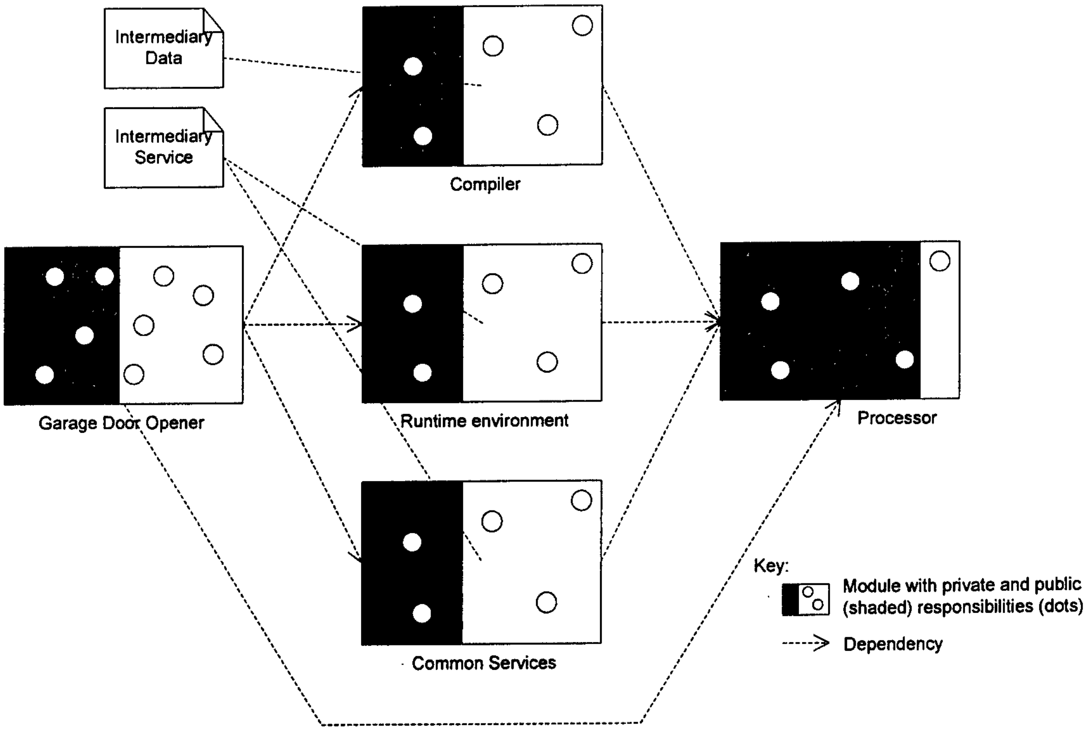


Figure 6: Allocation of Responsibilities According to the Applied Tactics

Changing the processor will affect the compiler and the runtime environment (which we can buy with the new processor), as well as the OS-like intermediary, which needs to be adapted to the new processor.

None of the responsibilities in the garage door opener modules should be affected by the processor change.

6 Coupling Concrete Scenarios to Design Fragments

Our goal in this report is to provide initial evidence that it is possible to derive design fragments from concrete scenarios. Our two example scenarios provided some limited evidence. In this section, we take a step back and treat the problem in general. That is, what is the general procedure for deriving design fragments from concrete scenarios? We have already seen some elements of this procedure (Steps 1-8). However, in addition to discussing these steps in general, we discuss the assets we assume will be available before any particular system is examined.

6.1 Assets Available

We assume that some set of assets is available prior to examining any particular set of concrete scenarios. These assets might be codified in an engineering handbook, embodied in a design assistant tool, and/or made available to the designer in some other form. These assets include

- *general-scenario-generation tables:* (see Appendix A) tables that provide valid "values" for each portion of a general scenario for various attributes (such as performance, modifiability, reliability, security, and usability)
- *mapping from a general scenario to reasoning frameworks:* A general scenario for a particular quality attribute contains enough information to identify some of the key parameters that must be reasoned about when designing an architecture to satisfy that attribute. By identifying those quality attribute parameters, the general scenario suggests reasoning frameworks (such as queuing theory) that might be applicable. At the most basic level, performance scenarios (which reference stimuli in terms of event arrival rates and response measures in terms of latency) suggest performance reasoning frameworks (such as scheduling theory or queuing theory); modifiability scenarios suggest modifiability reasoning frameworks, and so forth.

  A reasoning framework includes a list of independent parameters, a list of dependent parameters, and a means of deducing the value of a dependent variable when given the values of independent variables. Once a suitable reasoning framework is chosen, the goal is to determine which parameters have already been fixed and which parameters are still free. The free parameters become the focus of the architectural design activity. We first choose values for the free parameters and test them using the reasoning framework to see if quality attribute requirements are analytically satisfied. Then, we select architectural design fragments that are consistent with the chosen values.

- *tactics:* a list of tactics for various attributes. While we still have to work out the template for describing tactics, we expect each tactic to contain the following information:
  - references to one or more reasoning frameworks
  - the relationship between one or more model parameters and an attribute measure, and how the tactic acts to control those parameters
  - examples relating tactics to architectural design fragments
- *decision procedures:* a sequence of rules[4] that helps you choose a set of applicable tactics. We provide samplings of rules in this report. We expect these rules to include at least
  - identification/filtering questions: sets of questions that help to determine suitable tactics
  - optimization questions: known rules of thumb (expert knowledge) that can be used to shortcut the identification of tactics
  - "unfixing the fixed" questions: a set of questions to be used for reacting to seemingly untenable design situations to identify requirements that can be weakened

6.2 Steps for Coupling Concrete Scenarios to Design Fragments

We now briefly discuss the steps exemplified in our two scenarios.

1. *Pick a concrete scenario.* We are concerned, in general, with satisfying a set of requirements expressed in terms of concrete scenarios. We assume that this set of steps is a portion of a design method. This method will determine the concrete scenario that is to be treated next. In general, the scenario is being treated in the context of a number of design decisions that have already been made.
2. *Type-check concrete scenarios.* Each concrete scenario is an instance of a general scenario. Using the general-scenario-generation tables, we can verify that the concrete scenario is well formed by ensuring that it has all the elements of a general scenario. We also start identifying responsibilities from elements of the concrete scenario.
3. *Identify candidate reasoning frameworks.* One of the assets that we assume is available is an association between general scenarios and reasoning frameworks. We itemize possible reasoning frameworks. Some of the information from the concrete scenarios might

[4] The rules in this report are more like guidelines and heuristics; they certainly are not formal. However, we are hopeful that they provide a starting point for rules that will ultimately be codified for use in an expert system that serves as an architectural design assistant.

eliminate possible reasoning frameworks. For example, if we know that the response measure is a hard deadline, a queuing modeling framework is eliminated from consideration. Each reasoning framework has a collection of parameters that must be set before the reasoning framework can be applied.

4. *Determine the bound and free parameters.* The candidate reasoning framework has a number of parameters. Some of them might be given by the concrete scenarios, while others might be given by elements of the existing design that are unchangeable. For example, a concrete scenario may specify that "events arrive periodically." This specification may require a specific scheduling model. Another element of the existing design might be that a particular OS is to be used, which determines the execution time associated with processing one event. That execution time is a parameter of the model that is bound. All parameters not bound are considered to be free. The set of bound parameters reflects decisions that have *already* been made; the set of free parameters are decisions *yet* to be made.

   Dependent parameters are a special case. They are constrained by the response measure, not bound. Furthermore, all the tactics within a particular reasoning framework, by definition, affect the dependent parameters. Thus, dependent parameters are not considered in this determination.

5. *Determine the tactics associated with the free parameters.* One of the assets we assume is an enumeration of tactics and the parameters they control. This enumeration enables the tactics associated with the free parameters we use to be listed as candidate tactics for Steps 6-8.

6. *Assign the free parameters an initial set of values.* The designer makes an estimate for each free parameter based on intuition or knowledge. If the designer has no intuition or knowledge for a particular parameter, an arbitrary value can be chosen. If this parameter is important to the system, an implementation of a prototype might be appropriate to get an estimate.

   One question that comes up is "what if the initial set of values is radically incorrect?" We have three responses to this question:

   a. This is not a problem associated with our treatment of quality attributes. Any design method will require some knowledge of the parameters of importance. Otherwise, how can a design accommodate those quality attribute requirements?

   b. The values chosen, regardless of whether they are initial or final values, provide budgets or constraints for the remainder of the design. Decisions made during quality attribute analysis must be realized through the entire development process—architectural design, detailed design, and implementation.

   c. A sensitivity analysis can be done to show how sensitive the response measure is to uncertainties in the independent parameters. If there is no sensitivity, inaccuracies will not matter as much; if there is great sensitivity, the steps should lead to a better binding, in any case.

7. *Select tactics and apply them to bind values to achieve the required response measure.* In general, there will be multiple free parameters. We must determine bindings so that the response measure is satisfied. This is equivalent to determining satisfactory values for x and y in one linear equation in two unknowns. The values chosen for the two unknowns must be compatible with the equation and thus must be determined simultaneously, or at least the second value must be determined in the context of the first.

   We begin our description by considering the situation in which there is only one free parameter and one scenario. That is, we are considering a single concrete scenario in which there is only one free parameter and only one modeling framework.

   Each candidate tactic for that free parameter controls the parameter's value. For each candidate tactic, determine whether it can adjust the value of the free parameter to a new value where the resulting model's solution satisfies the response measure of the concrete parameter. If it can, the tactic becomes relevant. If it cannot, the tactic is discarded.

   Now consider multiple free parameters. In this situation, we need to consider simultaneously adjusting all free parameters. That is, if tactic 1 controls parameter 1, and tactic 2 controls parameter 2, we need to determine whether we can move the value for parameter 1 through tactic 1 and the value for parameter 2 through tactic 2, until the dependent parameter satisfies the response measure given by the concrete scenario. If we can, we have developed satisfactory bindings. If we have more than one tactic for each parameter, we need to consider all possible combinations of tactics for the parameters.

8. *Allocate responsibilities to the architectural elements of the design fragments that are associated with the selected tactics.* Every tactic applied during Step 7 has a design fragment associated with it and a set of rules that help to

   - Create/delete/refine architectural elements.
   - Add responsibilities to architectural elements.
   - Reallocate responsibilities of already-defined architectural elements.
   - Refine responsibilities and allocate them to architectural elements.

   For example, using the *Semantic-importance-based scheduling* tactic involves applying the following rules:

   - Create an architectural element "scheduler" with its associated responsibilities.
   - Allocate the responsibilities of higher importance to units of concurrency with a higher priority.

Using the *Break the dependency chain* tactic involves applying the following rules:

- Create an architectural element "intermediary."
- Add responsibilities to the intermediary that translate from the more abstract interface provided to the secondary modules to the concrete interface provided by the primary module.
- Refine the responsibilities of the secondary modules to use the services of the intermediary.

7 Summary and Remaining Open Issues

Our goal in this report was to present evidence that it is possible to move from a concrete scenario to a design fragment, while maintaining analytic capability. In the process, we wanted to identify open issues.

We demonstrated moving from a concrete scenario to a general scenario and discussed how to find tactics to achieve the desired response. This portion of our argument is the most thoroughly thought out. We then hypothesized associating a collection of design fragments with each tactic that both realizes the tactic and adheres to the model parameters that we have used to do the analysis. This association is clearly possible, although the list of design fragments may be rather long; selecting among them is an open issue.

Another open issue is the composition of design fragments in a methodical way, which is a topic of a future report. Composing two design fragments most likely changes parameters of the quality attribute models. For example, adding additional responsibilities to the intermediary to support modifiability may increase the execution time, which is a parameter in the performance model. Two important open issues in the composition area are (1) understanding the influences of design fragments on other quality attribute models and (2) modifying a composed design in such a way that the response measures continue to be satisfied.

We assume that the steps presented here are embedded in a design method. What are the characteristics of that method? In particular, in which order does it choose among the concrete scenarios? Does it treat one scenario at a time or multiple scenarios at a time? What does it do when the quality requirements (the concrete scenarios) over-constrain a solution?

Clearly this work is related to previous work on the Attribute-Driven Design method (ADD) [Bass 02]. ADD is a method for designing architectures that is based on isolating the "driving" quality requirements, identifying design decisions to satisfy them, and using those decisions as a constraint for further design decisions. Currently ADD provides no guidance on how to make design decisions satisfy the driving requirements. The steps discussed in this report and their envisioned extensions provide a basis for making such design decisions.

Finally, our steps depend heavily on the existence and general accuracy of quality attribute models. Such models exist for performance and are as accurate as the fidelity of their parameters to the values associated with an actual system. They exist much more weakly, if at all, for other quality attributes. Does this mean that our goal of providing analytic guidance to a designer is doomed to fail? We think not. First of all, if our design method is capable of providing analytic guidance for only performance, the architect is still advanced from the current state. Secondly, large communities are actively working on developing and refining quality attribute models, and some of these efforts should be successful in time. Lastly, we have seen decision procedures (at least in an initial form) that have the effect of causing an architect to focus on and think about a variety of factors that affect the quality attribute achievement of architectural design. Those decision procedures, by themselves, provide the architect with a checklist that will help to improve any design process.

Appendix A General Scenarios

Quality requirements of a system shape the software architecture of that system. To be useful, quality requirements should be specified precisely. General scenarios provide a vehicle for this. A quality attribute scenario consists of the following parts:

- a *stimulus*: a condition that needs to be considered when it arrives at a system
- a *response*: the activity undertaken after the arrival of the stimulus
- a *source of the stimulus*: the entity (e.g., a human or a computer system) that generated the stimulus
- an *environment*: the conditions under which the stimulus occurs; for example, when the system is in an overload condition
- a *stimulated artifact*: Some artifact is stimulated. It could be the whole system or pieces of it.
- a *response measure*: the attribute-specific constraint that must be satisfied by the response

The values (vocabulary) normally used to specify the six parts of a scenario depend on the quality that has to be achieved. We put the vocabulary used for the availability, performance, modifiability, usability, security, and testability qualities together into tables that can be used to generate general scenarios. The general-scenario-generation tables for modifiability and performance are shown in Table 14 and Table 15, respectively.

Table 14: Modifiability General-Scenario-Generation Table

Part of Scenario	Possible Values
source	• end user • developer • system administrator
stimulus	wants to add/delete/modify/change • functionality • quality attribute • capacity
environment	• runtime • compile time • build time • design time
artifact	• system • user interface • platform • environment • system that interoperates with target system
response	• locates places in the architecture to be modified • makes a modification without affecting other functionality • tests a modification • deploys a modification
response measure	• cost/effort in terms of – the number of components affected – effort – money • extent to which this affects other functions or quality attributes

Table 15: Performance General-Scenario-Generation Table

Part of Scenario	**Possible Values**
source	• one of a number of independent sources • possibly from within the system
stimulus	• periodic events arrive • sporadic events arrive • stochastic events arrive
environment	• normal conditions • overload conditions
artifact	• system • process
response	• processes stimuli • changes level of service
response measure	• latency • deadline • throughput • jitter • miss rate • data loss

General scenarios are quality attribute scenarios that are system independent—that is, they can be applied to any system. A general scenario can be generated by choosing several values—one from each column of the general-scenario-generation table—and combining them in a sentence or two. For example, a potential general scenario for performance is shown in Table 16.

Table 16: Example General Scenario for Performance

source	one of a number of independent sources
stimulus	periodic events arrive
environment	normal conditions
artifact	system
response	processes stimuli
response measure	latency

When written as two sentences, the scenario could be

> A periodic event from an independent source arrives at the system under normal conditions. The system has to process the stimulus within a certain latency.

Ultimately, a general scenario must be made system specific—called a *concrete scenario.* This conversion involves replacing the more generic vocabulary with concrete system-specific vocabulary. For example, the performance scenario shown above can be made system specific as follows:

> An event from sensor X arrives every 10 ms at a system that operates under normal conditions. The system has to process the stimulus within 1 ms.

Appendix B Tactics

We now briefly recapitulate our discussion of tactics from *Illuminating the Fundamental Contributors to Software Architecture Quality* [Bachmann 02]. A *tactic* is a means of satisfying a quality-attribute-response measure by manipulating some aspect of a quality attribute model through architectural design decisions.[5]

This definition of tactic has the following consequences:

- An architectural tactic is concerned with the relationship between design decisions and a quality attribute response. This response is usually something that would be specified as a requirement (e.g., an average latency requirement). Therefore, architectural tactics (by definition) are points of leverage for achieving quality attribute requirements. Consequently, codifying architectural tactics will involve articulating rules for making design decisions that control how and why the response varies.
- Analytic models for the various quality attributes allow us to identify design decisions that offer leverage for achieving quality attribute requirements. The analytic models also offer a reasoning framework for explaining how changes in the design decisions affect the response. Architectural tactics "live" within this reasoning framework.
- While architectural tactics are motivated from an analytic perspective, they have specific realizations in an architectural design. For example, while a queuing model might only require the average execution time as input to the model, there are many sources of execution time that need to be derived from an architectural design. Or, while a queuing model might be concerned only with how average latency varies as a function of the number of servers, in the architectural model, servers might be Web servers in one case and processors of a multiprocessor in another. There is a many-to-one relationship between the analytic model and the corresponding architectural realizations. Consequently, codifying architectural tactics involves articulating some of the possible mappings from an architectural model to an analytic, quality attribute model.

Performance Tactics

In general, latency is affected by the level and nature of the demand for resources. This includes event-arrival rates, the execution time resulting from event arrivals, and the variability level of each. Latency is also affected by the rules for choosing between conflicting demands for a resource. The inability to use a resource, either because it is being used to process other

[5] Note that this definition is different than in our previous writings [Bachmann 02, Bass 03].

events or is physically (e.g., due to failure) or logically (e.g., due to inadequate resource-allocation policies) unavailable, also affects latency. Therefore, we divide performance tactics into three high-level groups:

1. tactics for managing resource demand: These tactics control execution times and transfer times by controlling the demand for resources.
2. tactics for arbitrating between conflicting demands: These tactics control preemption and waiting times when competing requests take semantic importance or urgency (deadlines) into consideration and there is contention for shared resources.
3. tactics for managing multiple resources: These tactics enable multiple resources to be used efficiently to ensure that available resources are used when they are needed, thereby controlling the waiting time for them.

Tactics for Managing Demand

Event streams are the source of resource demand. Two stream traits characterize demand: the time between events in an event stream (i.e., the arrival rate) and how much of a resource is consumed by each request (a.k.a., execution or service time). The variances in arrival rates and execution times also affect latency.

The tactics for managing demand include

- *Manage the event rate.* Tactics in this category control how often events are generated.
- *Control the frequency of sampling external events.* If there is no direct control over the rate of externally generated events, queued events can be sampled at a lower frequency.
- *Reduce the computational overhead.* Computational work usually requires OS and middleware services to manage process interaction, communications, and the like.
- *Bound execution times.* This means placing a limit on how much execution time is used to respond to an event.
- *Bound queue sizes.* This tactic directly controls the maximum number of queued arrivals and effectively bounds arrival rates, but the consequence can be event loss.
- *Increase the computational efficiency of algorithms.* Improving the algorithms used in critical areas reduces the demand for processor time and therefore has the effect of improving latency.
- *Control the demand for resources.* When an algorithm requires the use of other resources, such as disks, its efficiency also depends on how those resources are used.

Tactics for Arbitrating Demand

When multiple streams make demands on the same (logical or physical) resource, tactics are needed for managing the shared resource. One of the main considerations for arbitration is the criterion used for selecting which competing streams should use the resource. Time-based, semantics-based, and fairness-based criteria are all considered. The fact that sometimes the resource is non-preemptable must also be considered. If such decisions are to be made at runtime, support for multiprocessing is helpful.

The tactics for arbitrating demand include

- *Increase the logical concurrency.* Mechanisms for achieving logical concurrency (such as processes and threads) allow the separation of concerns associated with processing event streams from the interleaving of such processing on a physical resource.
- *Determine the appropriate scheduling policy.* Many tactics are focused on the policies used to assign processor time to processes. The appropriate scheduling policy strongly depends on the system's goals. Scheduling tactics include
  - offline scheduling. A schedule can be constructed offline using assumptions about the time to be taken by each stream.
  - time-based scheduling. Informally, a "good" prioritization strategy takes into account the timing requirements associated with the stream (or even different arrivals within the same stream). This is reflected in three prioritization strategies:
    a. rate monotonic (RM) strategy—a static priority assignment for periodic streams that accords higher priorities to streams with shorter periods than to those with longer ones
    b. deadline monotonic (DM) strategy—a static priority assignment usually for periodic streams that accords higher priorities to streams with tighter deadlines than to those with more distant ones
    c. earliest deadline first (EDF) strategy—a dynamic priority assignment, usually for periodic streams, that accords higher priorities to process stimuli with tighter deadlines than to those with more distant ones
  - semantic-importance-based scheduling. The importance of different functions can vary by system mode. Therefore, the system must focus its resources on the appropriate work at the appropriate times.
  - aperiodic servers. Another tactic for increasing the overall utility of a system focuses on computational work that is normally relegated to background processing.
  - fairness-based scheduling. Several scheduling strategies that do not account for deadlines are first-in, first-out (FIFO) scheduling and processor sharing.
- *Use synchronization protocols.* When more than one thread of concurrency needs to access a shared resource (such as a queue or a data repository in a mutually exclusive manner) mechanisms such as locks and semaphores are commonly used. Various protocols include FIFO, priority inheritance, and priority ceiling protocols.

Tactics for Managing Multiple Resources

Exploiting the power of multiple resources to achieve performance goals introduces additional challenges to those managing single resources. However, physical concurrency brings with it the need to make resource-allocation decisions, such as which processor(s) should be used to process which event streams. When processing is physically distributed, the results computed on one processor are often needed by other processors, introducing the need for communication and synchronization. Making local copies can help reduce the overhead associated with moving data around.

The tactics for managing multiple resources include

- *Increase the physical concurrency.* A tactic for reducing latency is to increase the number of available resources. Additional processors, additional memory, and faster networks all have the potential for reducing latency by introducing the possibility of parallelism.
- *Balance resource allocation.* Appropriately allocating the load between multiple resources is important if physical concurrency is to be exploited maximally.
- *Maintain multiple copies of either data or computation.* Caching replicates data on different speed repositories or on separate repositories to reduce contention. The clients in a client-server pattern are replicated computations, also to reduce contention.

Modifiability Tactics

The tactics for affecting modifiability are organized into three sets: (1) those that localize expected modifications, (2) those that restrict the visibility of responsibilities, and (3) those that prevent the ripple effect. Each set is described below.

Tactics for Localizing Expected Modifications

The responsibilities that are assigned to modules greatly influence the cost of making a change. Depending on how the assignment was done, a specific change can affect either a single module or multiple ones. The goal of this set of tactics is to directly affect as few modules as possible with a single change by presenting guidelines for how responsibilities are assigned.

The tactics for localizing expected modifications include

- *Maintain semantic coherence.* Semantic coherence refers to the relationships between a module's responsibilities. Semantically coherent responsibilities are related by what they do, carrying out the same or at least similar functions. The goal here—ensuring that a module's responsibilities all work together without excessive reliance on other modules—is achieved by choosing responsibilities that have some sort of semantic coherence. Doing so binds responsibilities that are likely to be affected by a change.

- *Isolate the expected change.* Separating the responsibilities that are *likely* to change from those that are *unlikely* to change separates an architecture into fixed and variant parts. This enables more design effort to be devoted to making a selected subset of modules as easy to change as possible.
- *Raise the abstraction level.* Raising the abstraction level, and thus making a module more general, allows that module to calculate a broader range of functions based on input.
- *Limit options.* Limiting the set of possible modifications will reduce the variations that need to be considered in the design and simplify constructing a system suitable for modification. This tactic is in addition to those published in *Illuminating the Fundamental Contributors to Software Architecture Quality* [Bachmann 02].
- *Abstract common services in the primary modules.* Localize services that are used commonly by a variety of consumers. This tactic is in addition to those published in *Illuminating the Fundamental Contributors to Software Architecture Quality* [Bachmann 02].

Tactics for Restricting the Visibility of Responsibilities

If a module is affected by a change, it is important to know whether that change will become visible outside the module. If it will, changes to other modules will most likely be required.

The tactics for restricting visibility include

- *Hide information.* This tactic is based on dividing a module's responsibilities into two categories: public and private. Public responsibilities are those that are visible from both inside and outside the module. Private responsibilities are those that are visible only from inside the modules. The goal of this tactic is to limit the public responsibilities and make them visible through an interface.
- *Maintain existing interfaces.* This tactic is based on keeping interfaces constant across a particular change. That is, the module developer will maintain the old identity, syntax, and semantics of an existing interface even if the modification changes them.
- *Separate the interface from the implementation.* This tactic allows the realization of the implementation later in the development process than the interface specification.

Tactics for Preventing the Ripple Effect

A ripple effect from a modification is the necessity for making changes to modules that are not directly affected by that modification. This necessity occurs because of a dependency between the module that is directly affected and another module that is dependent on it.

The tactics for preventing the ripple effect include

- *Break the dependency chain.* This tactic refers to the use of an intermediary to keep one module from being dependent on another and therefore to break the dependency chain. Typically, the name of the intermediary depends on the type of dependency it breaks. The following examples of intermediaries can also be seen as tactics:
  - *Use a name server.* A name server breaks a dependency on the runtime location of a module.
  - *Use a virtual machine.* Virtual machines break dependencies on computations specific to a particular situation. Examples of virtual machines include the hardware abstraction layer and the Factory pattern [Gamma 95].
  - *Use a publish-subscribe pattern.* A publish-subscribe pattern breaks the dependency of a data consumer on the identity of the data producer.
  - *Use a repository.* A repository can be used to break two types of dependencies:
    1. the dependence of a data consumer on the identity of the publisher (as in the Publish-Subscribe pattern)
    2. the dependency of the data consumer on the data's syntax. Modern repositories allow the consumer to specify the type in which data is presented to them, regardless of the data's type in the repository.
  - *Use a dynamic-scheduling algorithm.* Some scheduling algorithms, such as semantic-importance-based scheduling, guarantee that deadlines will be achieved within certain restrictions.
- *Make the data self-identifying.* Tagging the data with identification information, such as sequence number (as in network protocols), syntax descriptions (as in some languages with dynamic runtime typing), or identity (as in free-form parameter invocation), will break dependencies on either sequencing or syntax.
- *Limit communication paths.* Restricting the modules with which a given module will communicate has the effect of ensuring that no dependency exists between the two modules. This tactic is in addition to those published in *Illuminating the Fundamental Contributors to Software Architecture Quality* [Bachmann 02].

References/Bibliography

[Bachmann 02] Bachmann, F.; Bass, L.; & Klein, M. *Illuminating the Fundamental Contributors to Software Architecture Quality* (CMU/SEI-2002-TR-025, ADA407778). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. <http://www.sei.cmu.edu/publications/documents/02.reports/02tr025.html>.

[Bachmann 00] Bachmann, F.; Bass, L.; & Klein, M. *Quality Attribute Design Primitives* (CMU/SEI-2000-TN-017, ADA392284). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. <http://www.sei.cmu.edu/publications/documents/00.reports/00tn017.html>.

[Bass 03] Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice*, 2nd ed. Reading, MA: Addison-Wesley, 2003.

[Bass 02] Bass, L.; Bachmann, F.; & Klein, M. "Quality Attribute Design Primitives and the Attribute-Driven Design Method," 169-186. *Proceedings of the 4th International Conference on Software Product Family Engineering*. Bilbao, Spain, October 3-5, 2001. Berlin, Germany: Spring-Verlag, 2002.

[Bass 01] Bass, L.; Klein, M.; & Moreno, G. *Applicability of General Scenarios to the Architecture Tradeoff Analysis Method* (CMU/SEI-2001-TR-014, ADA396098). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001. <http://www.sei.cmu.edu/publications/documents/01.reports/01tr014.html>.

[Gamma 95] Gamma, E.; Helm, R.; Johnson, R.; & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE March 2003	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE

Deriving Architectural Tactics: A Step Toward Methodical Architectural Design

5. FUNDING NUMBERS

F19628-00-C-0003

6. AUTHOR(S)

Felix Bachmann, Len Bass, Mark Klein

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER

CMU/SEI-2003-TR-004

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

HQ ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2116

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

ESC-TR-2003-004

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified/Unlimited, DTIC, NTIS

12B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

This is one of several reports that provide the current status on the work being done by the Software Engineering Institute (SEI^{SM}) to understand the relationship between quality requirements and architectural design. The ultimate objective of this work is to provide analysis-based guidance to designers so that the quality attributes of generated designs are more predictable and better understood.

Currently, four distinct problems must be solved to achieve that objective: (1) the precise specification of quality attribute requirements, (2) the enumeration of architectural decisions that can be used to achieve desired quality attribute requirements, (3) a means of coupling one quality attribute requirement to the relevant architectural decisions, and (4) a means of composing the relevant architectural decisions into a design. Embodying the solutions to these four problems into a design method that is sensitive to business priorities is an additional problem. This report deals with the third problem—coupling one quality attribute requirement to architectural decisions that achieve it.

This report provides initial evidence that there is, in fact, a systematic relationship between general scenarios, concrete scenarios, architectural tactics, and design fragments. It examines, in detail, two concrete scenarios—one for performance and one for modifiability—and describes how to move from each scenario, through tactics, to design fragments that satisfy the scenario.

14. SUBJECT TERMS

software architecture, software architecture design, software architectural tactics, quality attribute scenarios

15. NUMBER OF PAGES

66

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL